Hibernate

- **The purpose of Hibernate is to allow you to treat your database as if it stores Java objects.**
- However, databases in practice do not store objects—they store data in tables and columns.
- Unfortunately, there is no simple way to correlate the data stored in a database with the data represented by Java objects.

The difference between an object-oriented association and a relational one is fundamental. Consider a simple class to represent a user, and another to represent an e-mail address, as shown in Figure 5-1.

Figure 5-1. *An object-oriented association*

`User` objects contain fields referring to `Email` objects. The association has a direction—given a `User` object, you can determine its associated `Email` object. For example, consider Listing 5-1.

Listing 5-1. *Acquiring the* `Email` *Object from the* `User` *Object*

```
User user = ...
Email email = user.email;
```

The reverse, however, is not true. The natural way to represent this relationship in the database, as illustrated in Figure 5-2, is superficially similar.

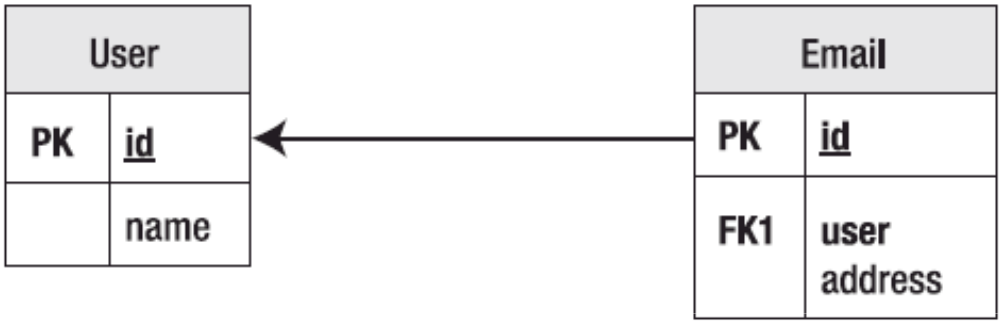

Figure 5-2. *A relational association*

Despite that similarity, the direction of the association is effectively reversed. Given an `Email` row, you can immediately determine which `User` row it belongs to in the database; this relationship is mandated by a foreign key constraint. It is possible to reverse the relationship in the database world through suitable use of SQL—another difference.

Given the differences between the two worlds, it is necessary to manually intervene to determine how your Java classes should be represented in database tables.

Why Mapping Cannot Be Automated

- It is not immediately obvious why you cannot create simple rules for storing your Java objects in the database so that they can be easily retrieved.
- For example, the most immediately obvious rule would be that a Java class must correlate to a single table.
- For example, instances of the User class could surely be represented by a simple table.

```
public class User
{
  String name;
  String password;
}
```

- And indeed it could, but some questions present themselves:
  - How many rows should you end up with if you save a user twice?
  - Are you allowed to save a user without a name?
  - Are you allowed to save a user without a password?

classes that refer to other classes

- When you start to think about classes that refer to other classes, there are additional questions to consider.
- Consider the Customer and Email classes defined below.

```
public class Customer {
  int customerId;
  int customerReference;
  String name;
  Email email;
}

public class Email {
   String address;
}
```

Based on this, the following questions arise:

- Is a unique customer identified by their customer ID, or their customer reference?
- Can an e-mail address be used by more than one customer?
- Should the relationship be represented in the `Customer` table?
- Should the relationship be represented in the `Email` table?
- Should the relationship be represented in some third (link) table?

Depending upon the answers to these questions, your database tables could vary considerably. You could take a stab at a reasonable design, such as that given in Figure 5-3, based upon your intuition about likely scenarios in the real world.

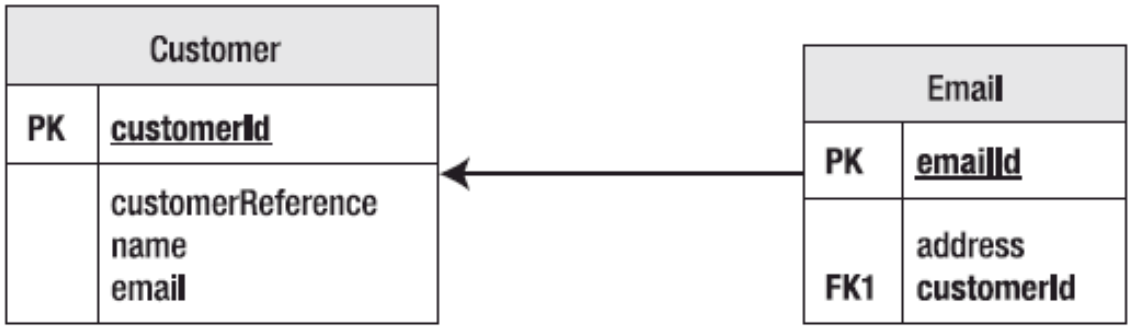

Figure 5-3. *Tables in which the customer is identified by* `customerId`*. Here, e-mail address entities can only be used by a single customer, and the relationship is maintained by the* `Email` *table.*

As soon as you take away the context provided by the variable and class names, it becomes much harder to form any useful decision about these classes.
It would be an impossible task to design an automated tool that could make this sort of decision.

Primary Keys

- Most "relational" databases that provide SQL access are prepared to accept tables that have no predefined primary key.
- Hibernate is not so tolerant—even if your table has been created without a primary key, Hibernate will require you to specify one.
- This often seems perverse to users who are familiar with SQL and databases, but who are not familiar with ORM tools.
- Without a primary key, it is impossible to uniquely identify a row in a table.

dminter	35
dminter	40
dminter	55
dminter	40
jlinwood	57

- This table clearly contains information about users and their respective ages.
- However, there are four users with the same.
- There is probably a way of distinguishing them somewhere else in the system—perhaps by an e-mail address or a user number.
- But if, for example, you want to know the ages of dminter with user ID 32, there is no way to obtain it from the table above.

While Hibernate will not let you omit the primary key, it will permit you to form the primary key from a collection of columns.

User	Usernumber	Age
dminter	1	35
dminter	2	40
dminter	3	55
dminter	32	42
jlinwood	1	57

- Neither User nor Usernumber contains unique entries, but in combination they uniquely identify the age of a particular user, and so they are acceptable to Hibernate as a primary key.
- **Why does Hibernate need to uniquely identify entries when SQL doesn't?**
  - Hibernate is representing Java objects, which are *always uniquely identifiable.*
  - *This is why the classic mistake* made by new Java developers is to compare strings using the == operator instead of the equals() method.
  - You can distinguish between references to two String objects that represent the same text and two references to the same String object.
- **SQL has no such obligation, and there are arguably cases in which it is desirable to give up the ability to make the distinction.**

If Hibernate could not uniquely identify an object with a primary key, then the following code could have several possible outcomes in the underlying table.

```
String customer = getCustomerFromHibernate("dcminter");
customer.setAge(10);
saveCustomerToHibernate(customer);
```

For example, let's say the table originally contained the data shown in Table 5-3.

Table 5-3. *Updating an Ambiguous Table*

User	Age
dcminter	30
dcminter	42

Which of the following should be contained in the resulting table?

- A single row for the user dcminter, with the age set to 10
- Two rows for the user, with both ages set to 10
- Two rows for the user, with one age set to 10 and the other to 42
- Two rows for the user, with one age set to 10 and the other to 30
- Three rows for the user, with one age set to 10 and the others to 30 and 42

In short, the Hibernate developers made a decision to enforce the use of primary keys when creating mappings so that this problem does not arise.

Lazy Loading

- When you load classes into memory from the database, you don't necessarily want *all the* information to *actually be loaded.*
- *To take an extreme example, loading a list of e-mails should* not cause the full body text and attachments of every e-mail to be loaded into memory.
  - First, they might demand more memory than is actually available.
  - Second, even if they fit, it could take a long time for all of this information to be obtained.
- If you were to tackle this problem in SQL, you would probably select a subset of the appropriate fields for the query to obtain the list; for example:

  SELECT from, to, date, subject FROM email WHERE username = 'someUser';
- Hibernate will allow you to fashion queries that are rather similar to this, but it also offers a more flexible approach, known as *lazy loading.*
- *Certain relationships can be marked as being* "lazy," and they will not be loaded from disk until they are actually required.

Classes and collections to be lazily loaded by default

- The default in Hibernate 3 is that classes (including collections like Set and Map) should be lazily loaded.
- For example, when an instance of the User class given in the next listing is loaded from the database, the only fields initialized will be userId and username.

```
public class User {
    int userId;
    String username;
    EmailAddress emailAddress;
    Set roles;
}
```

- However, as long as the object is still associated with Hibernate in the appropriate way, the appropriate objects for emailAddress and roles will be loaded from the database if they are accessed.
- This is the default behavior only; the mapping file can be used to specify which classes and fields should behave in this way.

Associations

```
public class Customer {
   int customerId;
   int customerReference;
   String name;
   Set email;
}
```

So, you should add another question: can a customer have more than one e-mail address? The set could contain a single entry, so you can't automatically infer that this is the case.

The key questions from the previous options are as follows:

- *Q1*: Can an e-mail address belong to more than one user?
- *Q2*: Can a customer have more than one e-mail address?

The answers to these questions can be formed into a truth table, as in Table 5-4.

Table 5-4. *Deciding the Cardinality of an Entity Relationship*

Q1 Answer	Q2 Answer	Relationship Between Customer and Email
No	No	One-to-one
Yes	No	One-to-many
No	Yes	Many-to-one
Yes	Yes	Many-to-many

These are the four ways in which the cardinality of the relationship between the objects can be expressed. Each relationship can then be represented within the mapping table(s) in various ways.

1. <one-to-one>

- The <one-to-one> element expresses the relationship between two classes, where each instance of the first class is related to a single instance of the second, and vice versa.
- Such a one-to-one relationship can be expressed either
  - By giving each of the respective tables the same primary key values, or
  - by using a foreign key constraint from one table onto a unique identifier column of the other.
- You would select a primary key association when you do not want an additional table column to relate the two entities.
- The master of the two entities takes a normal primary key generator, and its one-to-one mapping entry will typically have the attribute name and associated class specified only.
- The slave entity will be mapped similarly, but must have the constrained attribute setting applied to ensure that the relationship is recognized.
- Because the slave class's primary key must be identical to that allocated to the master, it is given the special id generator type of foreign.

- On the slave end, the <id> and <one-to-one> elements will therefore look like this:

```
<id name="id" column="product">
    <generator class="foreign">
    <param name="property">campaign</param>
    </generator>
</id>
<one-to-one name="campaign" class="com.hibernatebook.xmlmapping.Campaign" constrained="true"/>
```

<many-to-one>

- **The many-to-one association describes the relationship in which multiple instances of one class can reference a single instance of another class.**
- This enforces a relational rule for which the "many" class has a foreign key into the (usually primary) unique key of the "one" class.
- The following mapping illustrates the creation of a simple many-to-one association between a User class and an Email class.

```
<many-to-one name="email" class="com.hibernatebook.xmlmapping.Email"
                                                column="email" cascade="all" unique="true"/>
```

- The simplest approach to creating a many-to-one relationship, as shown in the previous example, requires two tables and a foreign key dependency.
- An alternative is to use a link table to combine the two entities. The link table contains the foreign keys referencing the two tables associated with both of the entities in the association.
- The following code shows the mapping of a many-to-one relationship via a link table.

```
<join table="link_email_user" inverse="true" optional="false">
<key column="user_id"/>
<many-to-one name="email" column="email_id" not-null="true"/>
</join>
```

- The disadvantage of the link table approach is its slightly poorer performance (it requires a join of three tables to retrieve the associations, rather than one).
- Its benefit is that it requires less extreme changes to the schema if the relationship is modified—typically, changes would be made to the link table, rather than to one of the entity tables.

COLLECTIONS AND ASSOCIATIONS

•The following code shows an implementation of mapping a set of strings into a property called titles:

```
<set name="titles" table="nameset">
<key column="titleid"/>
<element type="string" column="name" not-null="true"/>
</set>
```

•A typical implementation, however, maps other entities into the collection.

•Here we map Phone entities from the “many” side of a one-to-many association into a Set property, called phoneNumbers, that belongs to a User entity:

```
<set name="phoneNumbers">
<key column="aduser"/>
<one-to-many class="sample.Phone"/>
</set>
```

•If the Phone class contains a reference to a User object, it is not clear whether this constitutes a pair of unrelated associations or two halves of the same association—a bidirectional association.

When a bidirectional association is to be established, one side must be selected as the owner (in a one-to-many or many-to-one association, it must always be the “many” side), and the other will be marked as being the inverse half of the relationship.

The following code shows a mapping of a one-to-many relationship as a reverse association.

```
<set name="phoneNumbers" inverse="true">
<key column="aduser"/>
<one-to-many class="sample.Phone"/>
</set>
```

Unidirectional and bidirectional lists

If you need a real list to hold the position of the elements in a collection, you have to store that position in an additional column. For the one-to-many mapping, this also means you should change the `bids` property in the `Item` class to `List` and initialize the variable with an `ArrayList` (or keep the `Collection` interface from the previous section, if you don't want to expose this behavior to a client of the class).

The additional column that holds the position of a reference to a `Bid` instance is the `BID_POSITION`, in the mapping of `Item`:

```
<class name="Item"
       table="ITEM">
    ...
    <list name="bids">
        <key column="ITEM_ID"/>
        <list-index column="BID_POSITION"/>
        <one-to-many class="Bid"/>
    </list>

</class>
```

So far this seems straightforward; you've changed the collection mapping to `<list>` and added the `<list-index>` column `BID_POSITION` to the collection table (which in this case is the `BID` table). Verify this with the table shown in figure 7.6.

This mapping isn't really complete. Consider the ITEM_ID foreign key column: It's NOT NULL (a bid has to reference an item). The first problem is that you don't specify this constraint in the mapping. Also, because this mapping is unidirectional (the collection is noninverse), you have to assume that there is no opposite side mapped to the same foreign key column (where this constraint could be declared). You need to add a not-null="true" attribute to the <key> element of the collection mapping:

BID

BID_ID	ITEM_ID	BID_POSITION	AMOUNT	CREATED_ON
1	1	0	99.00	19.04.08 23:11
2	1	1	123.00	19.04.08 23:12
3	2	0	433.00	20.04.08 09:30

Figure 7.6
Storing the position of each bid in the list collection

```
<class name="Item"
       table="ITEM">
    ...
    <list name="bids">
        <key column="ITEM_ID" not-null="true"/>
        <list-index column="BID_POSITION"/>
        <one-to-many class="Bid"/>
    </list>

</class>
```

Note that the attribute has to be on the <key> mapping, not on a possible nested <column> element. Whenever you have a noninverse collection of entity references (most of the time a one-to-many with a list, map, or array) and the foreign key join column in the target table is not nullable, you need to tell Hibernate about this. Hibernate needs the hint to order INSERT and UPDATE statements correctly, to avoid a constraint violation.

Let's make this bidirectional with an `item` property of the `Bid`. If you follow the examples from earlier chapters, you might want to add a `<many-to-one>` on the `ITEM_ID` foreign key column to make this association bidirectional, and enable `inverse="true"` on the collection. Remember that Hibernate ignores the state of an inverse collection! This time, however, the collection contains information that is needed to update the database correctly: the position of its elements. If only the state of each `Bid` instance is considered for synchronization, and the collection is inverse and ignored, Hibernate has no value for the `BID_POSITION` column.

If you map a bidirectional one-to-many entity association with an indexed collection (this is also true for maps and arrays), you have to switch the inverse sides. You can't make an indexed collection `inverse="true"`. The collection becomes responsible for state synchronization, and the *one* side, the `Bid`, has to be made inverse. However, there is no `inverse="true"` for a many-to-one mapping so you need to simulate this attribute on a `<many-to-one>`:

```
<class name="Bid"
       table="BID">
    ...
    <many-to-one name="item"
                 column="ITEM_ID"
                 class="Item"
                 not-null="true"
                 insert="false"
                 update="false"/>

</class>
```

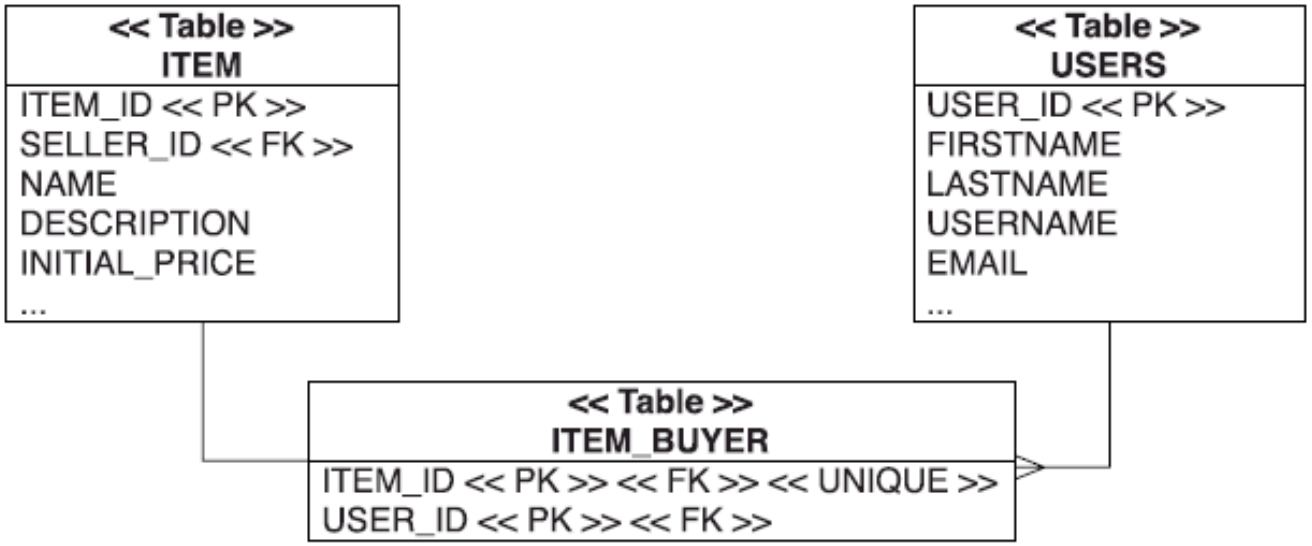

Figure 7.8 An optional relationship with a join table avoids nullable foreign key columns.

```
<set name="boughtItems" table="ITEM_BUYER">
    <key column="USER_ID"/>
    <many-to-many class="Item"
                  column="ITEM_ID"
                  unique="true"/>
</set>
```

You use a `Set` as the collection type. The collection table is the join table, `ITEM_BUYER`; its primary key is a composite of `USER_ID` and `ITEM_ID`. The new mapping element you haven't seen before is `<many-to-many>`; it's required because the regular `<one-to-many>` doesn't know anything about join tables. By forcing a `unique` constraint on the foreign key column that references the target entity table, you effectively force a one-to-many multiplicity.

You can map this association bidirectional with the buyer property of Item. Without the join table, you'd add a <many-to-one> with a BUYER_ID foreign key column in the ITEM table. With the join table, you have to move this foreign key column into the join table. This is possible with a <join> mapping:

```
<join table="ITEM_BUYER"
      optional="true"
      inverse="true">
     <key column="ITEM_ID" unique="true" not-null="true"/>
     <many-to-one name="buyer" column="USER_ID"/>
</join>
```

Two important details: First, the association is optional, and you tell Hibernate not to insert a row into the join table if the grouped properties (only one here, buyer) are null. Second, this is a bidirectional entity association. As always, one side has to be the inverse end. You've chosen the <join> to be inverse; Hibernate now uses the collection state to synchronize the database and ignores the state of the Item.buyer property. As long as your collection is not an indexed variation (a list, map, or array), you can reverse this by declaring the collection inverse="true". The Java code to create a link between a bought item and a user object is the same in both cases:

```
aUser.getBoughtItems().add(anItem);
anItem.setBuyer(aUser);
```

Many-to-many associations

The association between `Category` and `Item` is a many-to-many association, as can be seen in figure 7.9.

In a real system, you may not have a many-to-many association. Our experience is that there is almost always other information that must be attached to each link between associated instances (such as the date and time when an item was added

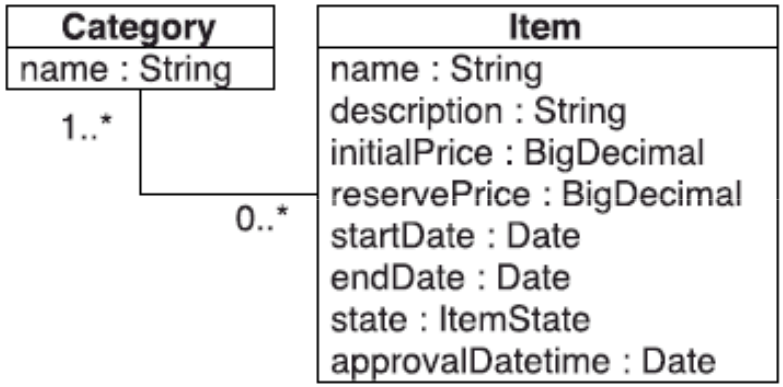

Figure 7.9
A many-to-many valued association between `Category` and `Item`

to a category) and that the best way to represent this information is via an intermediate *association class*. In Hibernate, you can map the association class as an entity and map two one-to-many associations for either side. Perhaps more conveniently, you can also map a composite element class, a technique we show later. It's the purpose of this section to implement a real many-to-many entity association.

A simple unidirectional many-to-many association

If you require only unidirectional navigation, the mapping is straightforward. Unidirectional many-to-many associations are essentially no more difficult than the collections of value-type instances we discussed earlier. For example, if the `Category` has a set of `Items`, you can create this mapping:

```
<set name="items"
     table="CATEGORY_ITEM"
     cascade="save-update">
    <key column="CATEGORY_ID"/>
    <many-to-many class="Item" column="ITEM_ID"/>
</set>
```

The join table (or *link table*, as some developers call it) has two columns: the foreign keys of the `CATEGORY` and `ITEM` tables. The primary key is a composite of both columns.

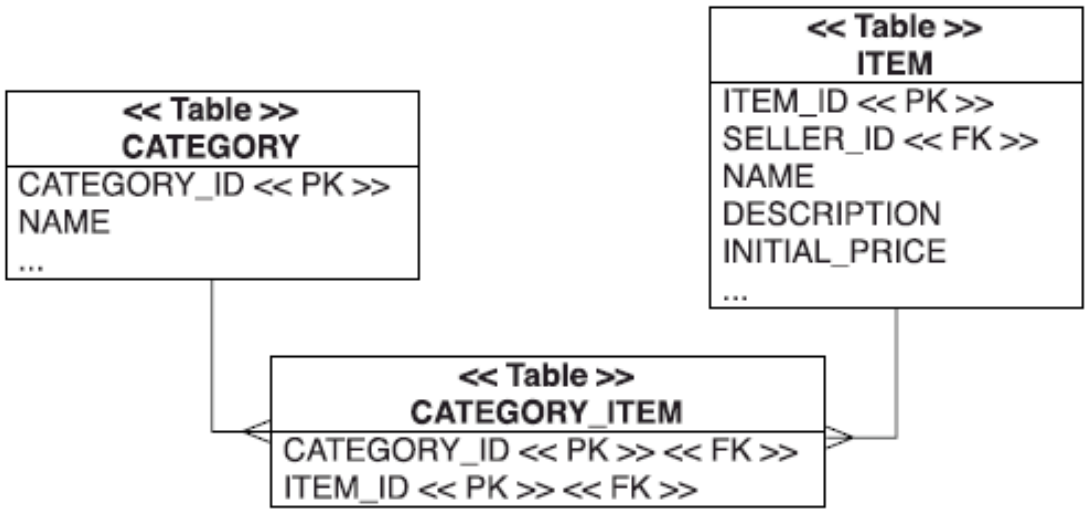

Figure 7.10 Many-to-many entity association mapped to an association table

In Hibernate XML you can also switch to an `<idbag>` with a separate primary key column on the join table:

```
<idbag name="items"
       table="CATEGORY_ITEM"
       cascade="save-update">
    <collection-id type="long" column="CATEGORY_ITEM_ID">
        <generator class="sequence"/>
    </collection-id>
    <key column="CATEGORY_ID"/>
    <many-to-many class="Item" column="ITEM_ID"/>
</idbag>
```

As usual with an `<idbag>` mapping, the primary key is a surrogate key column, `CATEGORY_ITEM_ID`. Duplicate links are therefore allowed; the same `Item` can be added twice to a `Category`.

A bidirectional many-to-many association

You know that one side in a bidirectional association has to be mapped as inverse because you have named the foreign key column(s) twice. The same principle applies to bidirectional many-to-many associations: Each row of the link table is represented by two collection elements, one element at each end of the association. An association between an `Item` and a `Category` is represented in memory by the `Item` instance in the `items` collection of the `Category`, but also by the `Category` instance in the `categories` collection of the `Item`.

Before we discuss the mapping of this bidirectional case, you have to be aware that the code to create the object association also changes:

```
aCategory.getItems().add(anItem);
anItem.getCategories().add(aCategory);
```

As always, a bidirectional association (no matter of what multiplicity) requires that you set both ends of the association.

When you map a bidirectional many-to-many association, you must declare one end of the association using `inverse="true"` to define which side's state is used to update the join table. You can choose which side should be inverse.

```
<class name="Category" table="CATEGORY">
    ...
    <set name="items"
         table="CATEGORY_ITEM"
         cascade="save-update">
        <key column="CATEGORY_ID"/>
        <many-to-many class="Item" column="ITEM_ID"/>
    </set>
```

You may reuse this mapping for the `Category` end of the bidirectional association and map the other side as follows:

```
<class name="Item" table="ITEM">
    ...
    <set name="categories"
         table="CATEGORY_ITEM"
         inverse="true"
         cascade="save-update">
        <key column="ITEM_ID"/>
        <many-to-many class="Category" column="CATEGORY_ID"/>
    </set>
</class>
```

Note the `inverse="true"`. Again, this setting tells Hibernate to ignore changes made to the `categories` collection and that the other end of the association, the `items` collection, is the representation that should be synchronized with the database if you link instances in Java code.

You have enabled `cascade="save-update"` for both ends of the collection. This isn't unreasonable, we suppose. On the other hand, the cascading options `all`, `delete`, and `delete-orphans` aren't meaningful for many-to-many associations. (This is good point to test if you understand entities and value types—try to come up with reasonable answers why these cascading options don't make sense for a many-to-many association.)